Desenvolupament de servei DNS intel·ligent

Enginyeria Tècnica en Informàtica de Sistemes
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)

Autor: Hugo Peris Bespin
Data: Setembre 2015

Director: Jordi Torres Viñals
Departament: Arquitectura de Computadors

President: David Carrera Pérez
Vocal: Maria Josefina Sierra Santibañez

Introducció

Scope del projecte al seu inici

Aquest projecte va sorgir de la idea d'un ex-alumne de la FIB de crear un servei DNS dinàmic per oferir-lo a empreses i organitzacions arreu del mon. La idea va arribar a en Jordi Torres qui va veure les necessitats i requeriments d'aquest projecte i va decidir que seria bo partir-lo en dos peces.

Per una banda el propi servei DNS intel·ligent i per una altra un sistema de monitorització que alimentés al DNS intel·ligent.

En aquest PFC es tracta la part del servei de DNS intel·ligent.

Als requeriments oficials del projecte es comptava també amb una part de gestió i configuració del servei de DNS per als clients. Aquesta part corresponia a una interfície web on es podrien configurar els registres DNS i la monitorització per als *websites* dels clients. Això seria dut a terme per una tercera persona.

Desenvolupament del projecte en mode col·laboratiu

Una vegada clares les tres àrees de desenvolupament del projecte ens vam reunir tots els implicats, inclosos en Jordi Torres per definir l'abast de tot el projecte i un esborrany de l'arquitectura i tecnologies a fer servir:

- pàgina web per la gestió del servei (Java i Cassandra)
- sistema de monitorització (Ruby on rails)
- servei DNS intel·ligent (Java)

Per una altra banda la resta de membres de l'equip s'encarregarien de les tasques comercials i administratives. Ja que estaven en una bona posició per obtenir clients i donar a conèixer el producte. I també perquè al venir la idea d'una persona actualment desenvolupant un negoci es podia aprofitar aquest coneixement quan arribés el moment de comercialitzar el servei.

A partir d'aquest punt es van anar mantenint reunions de forma periòdica per veure com evolucionava el desenvolupament de cadascuna de les parts, introduir millores i funcionalitats, i identificar bloquejos.

Durant tot aquest temps el procés de desenvolupament de les diferents peces es va fer per separat esperant que al moment adequat, quan el software estigués més madur, s'integressin les peces entre si.

Es treballava a uns servers comuns ubicats a Amazon, i amb un control de versions amb Subversion. El software tot i així es desenvolupava i provava en local, deixant versions estables corrent en els servidors de proves.

Pausa del projecte

Arribats a un punt on el software de DNS intel·ligent estava molt evolucionat, encara que no suficientment madur per integrar-se amb les altres peces, van sorgir assumptes d'índole personal que van fer que no es pogués treballar al projecte amb la dedicació necessària.

Fins al punt que es va parar el desenvolupament. La motivació inicial pel projecte de crear un producte i una empresa per vendre'l havia desaparegut. Havia deixat de ser prioritari i després de parlar-ho amb la resta de membres de l'equip es va decidir cancel·lar el projecte.

Així que arribem a una fase de pausa del projecte.

Represa del projecte en solitari

Passat un temps, concretament 4 anys, decideixo reprendre la meva part del projecte: el software de DNS intel·ligent. Ja fa molts anys que vaig acabar les classes, i només em queda el PFC per completar la carrera.

Un nou pla d'estudis es farà efectiu en poc temps, i el meu actual passarà a ser invàlid. En la convalidació al nou pla passaré a tenir molt més crèdits pendents que ara mateix, no només el PFC.

Amb aquesta nova motivació rescato el projecte. Un gran repte m'espera per davant: trobar els backups i tot l'històric del treball fet, entendre que i com vaig desenvolupar en aquell moment, analitzar fins a quin punt les funcionalitats estaven llestes, arreglar bugs al codi, crear noves funcionalitats i deixar llesta una versió del software estable i funcional.

S'ha de tenir present que donat que aquest projecte es va iniciar al 2010 hi ha certes decisions que s'han d'entendre en el context d'aquell moment. Es van prendre decisions basant-nos en la experiència dels involucrats anys enrere, i de la penetració d'algunes tecnologies a la comunitat i empreses. També que certes funcionalitats que podien ser molt noves llavors ara ja estan molt assimilades, proveïdors DNS d'internet, i fabricants les ofereixen als seus productes.

Objectius del projecte

En aquesta nova etapa del PFC els objectius en quant al que ha de fer el software de DNS intel·ligent són clars i depenen exclusivament mi.

Es vol tenir un servei de DNS capaç de resoldre els registres habituals: A, CNAME, NS, MX amb possibilitat de round-robin pels registres que ho admetin i els que no, com el CNAME per exemple, amb un únic valor possible.

Es vol implementar el motor de DNS intel·ligent sobre els registres DNS clàssics, de la següent manera:

- Volem ser capaços de configurar registres DNS molt semblants a la manera clàssica (sense lògica afegida). Aquests registres els anomenarem **estàtics**.

  L'avantatja que tindran respecte a un registre DNS clàssic serà que podrem configurar múltiples entrades A per un nom i també CNAME (això no està suportat pels serveis DNS clàssics).

- Volem per un mateix nom poder resoldre un valor o un altre en funció de l'hora del dia en la que ens trobem. Aquests registres els anomenarem **dinàmics per franja horària**.

- Volem per un mateix nom poder resoldre un valor o un altre en funció de la localització geogràfica de la IP que ens ha enviat la petició. Aquests registres els anomenarem **dinàmics per GEO localització**.

- Volem per un mateix nom poder resoldre un valor o un altre en funció de la xarxa o IP específica que ens ha enviat la petició. Aquest registres els anomenarem **dinàmics per subnetting**.

Volem portar el desenvolupament amb el seus canvis registrat a un sistema de control de versions (acrònim en anglès: VCS, de *Version Control System*). De manera que en cada moment sigui possible saber quan es va introduir un canvi i tornar enrere totes les revisions que facin falta si en algun moment es detecta un error gran al codi.

Es vol muntar una petita infraestructura per oferir el servei a Internet i configurar els hosts i serveis mitjançant una eina d'administració de configuracions. Així es podran distribuir els canvis a tota la plataforma de manera ràpida i centralitzada. També es vol que l'evolució d'aquesta configuració estigui registrada al VCS.

Tota aquesta infraestructura es vol crear a un proveïdor de Cloud Computing per comoditat i flexibilitat. I es volen tenir backups del VCS a un altre proveïdor de Cloud per si hagués un desastre poder recrear la plataforma de nou.

D'aquesta llista d'objectius han caigut varis que no depenien 100% de mi en els inicis del projecte (en l'etapa de desenvolupament col·laboratiu).

- S'ha perdut la possibilitat de per un mateix nom retornar un valor o un altre en funció de si el valor a retornar està realment disponible.

  Aquesta part del projecte es la que s'havia emmarcat en un altre PFC independent donada la seva complexitat i carrega de feina que portava.

  Es volien crear múltiples possibilitats de monitors i que hagués un sistema que estigués constantment testejant-los per tenir informació el més acurada possible de si estem retornant un valor correcte. Amb això es pretenia evitar resoldre un nom o IP d'un host que estigués caigut.

- Tampoc existeix la possibilitat d'editar la configuració, afegir dominis, etc. a través d'una interfície web. Aquest part es la que s'encarregava de fer una tercera persona col·laboradora de projecte.

  Tenia molt de sentit fer-la llavors ja que es pretenia oferir el servei a empreses perquè es configuressin els seus registres dinàmics i la monitorització que desitjaven. Però en aquest moment del projecte el principal objectiu es només tenir un servei que ofereixi les funcionalitats que s'han descrit abans.

  Per tant el servei de DNS intel·ligent es configurarà directament al host on estiguem instal·lant el software com si es tractés d'un altre servei més del sistema.

Implementació del DNS

Sobre quin llenguatge desenvolupar?

Una de les primeres preguntes que es plantegen quan es vol desenvolupar es sobre quin llenguatge fer-ho.

En aquest cas quan es va iniciar el projecte es van valorar les opcions més conegudes en aquell moment (recordem: 2010), i que podien funcionar millor per crear un servei que hagués de córrer 24h resolen peticions cara a internet. Aquestes opcions es van enfrontar amb el meu coneixement i experiència en aquests llenguatges.

Es va pensar en C++ i Java principalment. S'ha de dir que jo, personalment, no soc programador experimentat, i a la meva vida laboral el que programo ho faig principalment en llenguatges interpretats (Perl i Php en aquella època).

Es volia fer servir un llenguatge compilat (com C++ o Java) ja que són més robustos a més de més eficients, i per oferir un servei DNS 24/7 a Internet això es primordial, sobretot la robustesa respecte llenguatges interpretats.

Així que a l'hora d'escollir un llenguatge s'havia de trobar un que em resultés familiar. La meva experiència en aquests llenguatges es basava principalment en l'ús que havia fet d'ells durant les diferents etapes a la FIB, en diferents assignatures. La majoria de vegades que en una assignatura havíem de programar alguna cosa ho fèiem en Java o en C/C++. Durant la carrera també vam programar en altres llenguatges com per exemple ASM (assemblador), personalment m'encanta però no tenia cabuda per un projecte així la veritat. En resum, principalment l'experiència en basava en aquests dos grans: C++ i Java.

Una vegada amb dos finalistes es van valorar les avantatges i inconvenients d'un i l'altre. Principalment:

- **C++**
  pros: més eficient
  cons: més complexitat per compilar i una vegada compilat no portable entre plataformes

- **Java**
  pros: facilitat per compilar, desenvolupar, i portable entre plataformes una vegada compilat
  cons: més ineficient

Es va donar prioritat a la portabilitat entre plataformes i es va escollir **Java**. Ja que facilitava la posta en marxa i expansió del software al poder-se independitzar de la plataforma.

Això a la llarga s'ha vist que va ser un encert ja que durant l'evolució i desenvolupament del projecte s'ha treballat en Windows, Linux i MacOS sense cap tipus de incompatibilitat. A més la comunitat en Java és molt gran, hi ha molta documentació, així que és fàcil tirar endavant quan et trobes amb un blocker. Programar en C++ en aquest sentit hagués estat més complicat i hagués requerit més temps.

Objectiu: No reinventar la roda

Després d'haver escollit el llenguatge de programació la següent qüestió va ser plantejar-se el no reinventar la roda. Si en el mon Java ja existien servidors DNS fiables que responien a les necessitats bàsiques del que s'espera d'un servei de DNS, es podia treballar sobre aquesta base per ampliar-la amb les noves funcionalitats.

Investigant les solucions disponibles al 2010, es va veure que l'opció més factible era *EagleDNS* (http://www.unlogic.se/projects/eagledns), basat en les llibreries de *dnsjava* (http://www.dnsjava.org/).

EagleDNS oferia el servei DNS en Java amb fitxers de configuració típics a l'estil de *Bind*. A efectes pràctics un fitxer de configuració en *Bind* era perfectament compatible amb *EagleDNS*. Això ajudava també a entendre com funcionava per dins *EagleDNS* ja que per la meva experiència laboral tenia perfectament assimilat el format del fitxer de configuració de zones de *Bind*.

Una ràpida ullada al codi i unes proves van servir per determinar que l'elecció era bona. El codi era prou llegible per entendre com funcionava la càrrega del registres, la gestió de peticions (entrada sortida de paquets) i la part més interesant: la resolució. Era en aquest punt on es podia introduir el nou motor de resolució que tingués la lògica pels registres intel·ligents que es volien implementar.

La versió d'*EagleDNS* sobre la que treballar va la ser 1.0. A més d'això feia falta una llibreria extra també desenvolupada per la gent d'*EagleDNS* (*unlogic.se*), anomenada *StandardUtils*.

Aquesta llibreria, o paquet de software, conté alguns mètodes d'indole genèric que són necessaris per fer funcionar *EagleDNS*, per exemple utilitats per llegir fitxers. Estan ubicats fora del propi codi del servidor DNS ja que en els altres projectes de *unlogic.se* els fan servir també. Així s'aprofita el codi ja programat.

Anàlisis del codi existent

Una vegada decidida la base amb la que es treballaria toca analitzar el codi existent per:

- Primer, entendre com funciona sense cap tipus de modificació
- Segon, començar pensar en com i a on introduiríem les modificacions que derivaran les peticions al nou motor DNS

EagleDNS funciona de la següent manera, des de la classe principal:

- Crea un tipus d'objecte que llegirà els fitxers de configuració i guardarà el contingut de les zones amb el seus registres.

```
private final HashMap<String, ZoneProvider> zoneProviders
```

- Crea uns tipus d'objectes que s'encarreguen de gestionar la comunicació amb els clients a nivell TCP i UDP

```
private ArrayList<TCPSocketMonitor> tcpMonitorThreads
private ArrayList<UDPSocketMonitor> udpMonitorThreads
```

La configuració dins de l'objecte *zoneProviders* es carrega de la següent manera, per cada proveïdor de zones (en aquest cas hi ha només un: els fitxers), s'examinen les zones que hi ha, això es fa cridant al mètode *getPrimaryZones* ubicat a la classe *FileZoneProvider*

```
for (Entry<String, ZoneProvider> zoneProviderEntry : this.zoneProviders.entrySet()) {
  primaryZones = zoneProviderEntry.getValue().getPrimaryZones();
```

Una vegada dins de la classe *FileZoneProvider* :

- S'obtenen les zones llistant els fitxers que hi ha al directori de configuració (variable *allFiles*).
- Per cada fitxer s'obté el nom de la zona (és el mateix que el nom del fitxer).
- Es crea un objecte nou de tipus *Zone* (aquest tipus d'objecte es troba definit dins de les classes del paquet de *dnsjava*).
- Es passa aquest objecte *Zone* al mètode *add* que és el s'encarrega de fer el parsing de fitxers.

```
public Collection<Zone> getPrimaryZones() {
  for(File zoneFile : allFiles){
    origin = Name.fromString(zoneFile.getName(), Name.root);
    Zone zone = new Zone(origin, zoneFile.getPath());
    zones.add(zone);
```

Arribats a aquest punt ja tenim totes les zones carregades amb els seus registres. Per tant podem passar a servir peticions.

Les peticions es processen de la següent manera, en el moment que es rep una es passa a la funció *generateReply* de la classe principal d'*EagleDNS* que comprova si és una petició DNS vàlida.

```
byte[] generateReply (Message query, byte[] in, int length, Socket socket) throws
IOException {
  header = query.getHeader();

  if (header.getFlag(Flags.QR)) {
    return null;
  }
  if (header.getRcode() != Rcode.NOERROR) {
    return errorMessage(query, Rcode.FORMERR);
  }
  if (header.getOpcode() != Opcode.QUERY) {
    return errorMessage(query, Rcode.NOTIMP);
  }
  if (!Type.isRR(type) && type != Type.ANY) {
    return errorMessage(query, Rcode.NOTIMP);
  }
```

Si la petició es vàlida la passa al mètode encarregat de resoldre peticions, anomenat *addAnswer*, és aquí on *EagleDNS* comprova si té alguna zona on podria estar el registre amb el nom que se li ha preguntat.

```
byte rcode = addAnswer(response, name, type, dclass, 0, flags, IP);
```

Arribats a aquest punt ja tenim el codi proporcionat per EagleDNS analitzat en el que necessitem nosaltres:

- Entrada de dades
- Resolució de peticions

Per tant sembla correcte pensar que s'ha d'intervenir en aquest dos punts. Per una banda tenim que carregar el registres intel·ligents, i per una altra tenim que poder-los resoldre quan arriba una petició.

Afegir noves funcionalitats al codi actual: Camelia DNS

De l'anàlisi de l'apartat anterior ja tenim una idea de com introduirem els canvis que es volen per afegir les noves funcionalitats.

Però abans de continuar hi ha una part important que encara no s'ha tractat. Aquestes noves funcionalitats, aquest nou software, quin nom tindrà? No volia que fos un conjunt de classes que s'anomenés simplement DNSintel-ligent, o DNS++, o quelcom així molt impersonal.

Vaig decidir triar el nom de **Camèlia**, és el nom d'una flor. Aquest nom té un raó, i és que pretenia que fos un petit homenatge al negoci que des de fa molt anys tenia la meva família a Barcelona i que es va haver de tancar al 2006. El nom final simplificat sense accent i afegint el sufix DNS es **CameliaDNS**.

Amb un nom decidit pel nou DNS es procedeix a afegir els canvis pertinents sobre el software d'*EagleDNS*.

Per una banda a la classe principal d'*EagleDNS* es declara

- Un objecte de tipus *CameliaDNS*.
- Un objecte de tipus *GeoIP* per resoldre el registres de GEO Localització

```
private CameliaDNS Camelia = new CameliaDNS();
private LookupService GeoIP;
```

Amb això només hem declarat els objectes, encara no estan inicialitzats. La inicialització la fem en el moment del procés de carrega de les zones de la següent manera. Al mateix punt on a l'anàlisi de l'apartat anterior hem vist que es crida a la carrega de zones afegim aquestes crides:

```
this.Camelia.init();
this.GeoIP = new LookupService("geoip/GeoIP.dat",...);
```

I per una altra banda al mètode *addAnswer* de la classe principal d'*EagleDNS* afegim una crida per resoldre la petició DNS contra l'objecte de tipus *CameliaDNS* declarat abans.

```
private byte addAnswer(...,InetAddress IP) {
  RRset records = this.Camelia.resolveQuery(name,IP,this.GeoIP);
```

A més a més també s'ha afegit un paràmetre de tipus *InetAddress* a la capçalera del mètode *addAnswer* ja que es necessari pel mètode de resolució de *CameliaDNS*.

Aquesta modificació es necessària des del moment que volem que el nou motor de resolució sigui capaç d'entregar un resultat o un altre en funció de la IP que ens ha enviat la petició. A la implementació base d'*EagleDNS* la IP es podia quedar a la capa de comunicació ja que era una dada inútil pel mètode de resolució. Però en canvi *CameliaDNS* necessita que se la proporcionin per que alguns dels seus mètodes de resolució estan basats en això.

Que és CameliaDNS?

Com ja s'ha explicat *CameliaDNS* és un paquet de classes Java que expandeix funcionalitats al servidor de DNS *EagleDNS*, servidor de DNS de tipus clàssic. Aquestes noves funcionalitats s'han de carregar després de fer unes modificacions a *EagleDNS* per tal de que aquest sigui capaç d'aprofitar-se del nou software.

CameliaDNS està format principalment per:

- Fitxers de configuració
- Classes Java, de les quals els objectes principals de CameliaDNS són:
    - CameliaZone
    - CameliaRecord
    - CameliaPolicy

Fitxer de configuració Camelia.xml

La configuració de CameliaDNS es troba en un fitxer, però això no va ser exactament així al inici del projecte al 2010. Com es pretenia tenir múltiples clients entrant a la plataforma web per actualitzar el seus registres es va pensar en que la web guardés aquestes configuracions en algun tipus de base de dades.

Es va optar per una solució no-SQL amb Cassandra. La web guardaria les configuracions a l'estructura de dades de Cassandra, CameliaDNS hauria de venir a buscar-les aquí, i comprovar que no hagués actualitzacions. A la seva vegada Cassandra també es faria servir per emmagatzemar informació dels monitors del segon PFC implicat, i CameliaDNS consultaria aquestes dades per saber si un host es trobava disponible o no abans de retornar el resultat.

Com també es volia tenir la possibilitat de carregar configuracions des de fitxers, es va començar a desenvolupar primer aquesta via ja que era més senzilla i permetia ser més independent per tirar endavant CameliaDNS. Cassandra era sobretot necessari per la integració amb les altres parts del projecte. A més de cara al desenvolupament de la lògica del motor de CameliaDNS a nivell funcional no canviava el entregar un resultat que s'havia configurat des d'un fitxer o des d'una base de dades, així que els esforços es van concentrar en desenvolupar i evolucionar el motor de CameliaDNS.

Finalment degut a l'aturada del projecte, i la represa en solitari del PFC, no es va seguir evolucionant la via de carregar la configuració de Cassandra. Des de fitxer ja es compleix amb els objectius marcats ara pel PFC.

Per tant ara mateix CameliaDNS té només una única possibilitat de configurar zones, amb fitxers. Concretament fitxers XML, aquest fitxers i la seva ubicació han de seguir les següents regles.

Nomenclatura pel noms de fitxers de configuració i ubicació d'aquests

- Els fitxers de configuració s'han d'ubicar dins del directori *cameliaZones*. Aquest directori ha d'estar ubicat dins del directori arrel *d'EagleDNS*, al mateix nivell que el directori de configuració *conf d'EagleDNS*

- El nom del fitxer marca el nom de la zona. Si volem configurar una zona pel domini *cameliadns.net* hem de crear el fitxer com *cameliadns.net.xml*

Format del contingut del fitxer de configuració XML

Pel que respecte al contingut del XML, ha de tenir el següent format quan es configuren les zones, registres bàsics, i registres de tipus CameliaDNS

```
<?xml version="1.0" encoding="UTF-8"?>
<Domain xmlns="http://www.cameliadns.net"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <DomName>cameliadns.net.</DomName>
    <UfName>cameliadns.net</UfName>
    <Bind>
        <PrimaryDNS>ns1.cameliadns.net.</PrimaryDNS>
        <Email>admin.cameliadns.net.</Email>
        <Serial>30</Serial>
        <NS>
            <Value>cameliadns.net.</Value>
        </NS>
        <A>
            <Value>72.21.206.6</Value>
        </A>
        <MX>
            <Value>mx.cameliadns.net.</Value>
            <Priority>10</Priority>
        </MX>
    </Bind>
    <Camelia>
        <Record>
            <Name>email</Name>
            <RecordTTL>90</RecordTTL>
            <Policies>
                ...
            </Policies>
        </Record>
        <Record>
            <Name>www</Name>
            <RecordTTL>30</RecordTTL>
            <Policies>
                ...
            </Policies>
        </Record>
    </Camelia>
</Domain>
```

** els punts suspensius dins del tags *Policies* són deguts a que s'explicarà més endavant. Ara no es vol distreure l'atenció del que és absolutament necessari

- Els camps *Domain*, *DomName* i *UfName* han de coincidir i ser coherents amb el nom del fitxer i nom de domini que es desitja configurar.

- Hi ha dos seccions obligatòries

    - *Bind*: per la configuració bàsica de la zona com els registres NS, MX, A del root del domini, email i serial (no hi ha necessitat d'actualitzar-lo en cada edició).

    - *Camelia*: per la configuració dels registres de CameliaDNS

- A la secció *Camelia* hi hauran tantes entrades *Record* com registres volem configurar

- Per cada *Record* es necessari definir:

    - *Name*: únic i irrepetible en forma relativa a la zona

    - *TTL*: temps d'expiració del registres a las caches de proveïdors d'internet, clients de DNS de usuaris i servidors, etc.

    - *Policies*: les policies s'expliquen més endavant, són els diferents tipus de registres intel·ligents que ofereix CameliaDNS. Només pot haver una definida

CameliaZone(s), que són?

Una *CameliaZone* és un objecte que es crea en temps d'execució de les classes Java de *CameliaDNS* que conté la configuració d'una zona que li ha vingut donada pel fitxer de configuració corresponent.

Durant l'execució de *CameliaDNS* pot haver tantes *CameliaZones* com zones hagin definides en fitxers de configuració.

No pot haver dos *CameliaZones* iguals ja que cadascuna d'elles representa una zona, i no poden haver dos zones iguals.

Per cada objecte de tipus *CameliaDNS* hi ha un objecte de tipus *HashMap* de *String* +*CameliaZone* que conté totes les *CameliaZones* que existeixen definides en fitxers de configuració. Aquest *HashMap* es defineix a nivell global en la classe principal de *CameliaDNS* de la següent manera:

```
private HashMap<String,CameliaZone> zoneMap = new HashMap<String,CameliaZone>();
```

Mètodes de CameliaZone

Hi ha els següents mètodes disponibles pel tipus CameliaZone

- Constructor públic CameliaZone

  ```
  public CameliaZone(String zone, File cameliaDomainConfigFile)
  ```

  Crea un objecte del tipus *CameliaZone* a partir d'un *String* que defineix el nom de la zona i un *File* que és el fitxer on es pot trobar la configuració de la zona.

- getZone

  ```
  public String getZone()
  ```

  Retorna el nom de la zona en format *String*

- getZoneName

  ```
  public Name getZoneName()
  ```

  Retorna el nom de la zona en format *Name* (tipus definit al paquet *dnsjava*)

- findRecord

  ```
  public CameliaRecord findRecord(Name name)
  ```

  Busca si un *Name* passat per paràmetre es troba dins dels registres d'aquesta zona. Si el troba el retorna, si no el troba retorna un registre buit.

- Funció privada loadCameliaDNSConfig

  ```
  private void loadCameliaDNSConfig ()
  ```

  Llegeix del fitxer la configuració dels registres de *CameliaDNS* i la carrega al objecte *records*.

- Funció privada debugRecords

  ```
  private void debugRecords ()
  ```

  Imprimeix en nivell debug tots el registres que hi ha per aquesta zona

CameliaRecord(s), que són?

Un *CameliaRecord* és un objecte que es crea en temps d'execució de les classes Java de *CameliaDNS* que conté la configuració d'un registre de *CameliaDNS* especificada al fitxer de configuració de la zona a la qual pertany.

Durant l'execució de *CameliaDNS* pot haver tants *CameliaRecords* com registres hagin definits en fitxers de configuració.

No pot haver dos *CameliaRecords* iguals ja que cadascun d'ells representa un registre DNS amb el seu domini, o zona, com a sufix, és a dir són objectes que representen noms FQDN (Full Qualified Domain Name).

Cada objecte de tipus *CameliaZone* conté un objecte de tipus *ArrayList* de *CameliaRecord* que conté tots els registres *CameliaRecords* que existeixen definits dins del fitxer de configuració d'aquesta *CameliaZone*. Aquest *ArrayList* es defineix a la classe *CameliaZone* de la següent manera:

```
private ArrayList<CameliaRecord> records = new ArrayList<CameliaRecord>();
```

Mètodes de CameliaRecord

- Constructor públic CameliaRecord

  ```
  public CameliaRecord(String absoluteZone, XMLSettingNode cameliaConfigXMLSettings)
  ```

  Crea un objecte del tipus *CameliaRecord* a partir d'un *String* que defineix el nom de la zona on es troba aquest *CameliaRecord* i un *XMLSettingNode* que és la part del fitxer XML de la zona que conté la configuració d'aquest registre.

- isNull

  ```
  public boolean isNull()
  ```

  Retorna un booleà indicant si el *CameliaRecord* es null o no

- getName

  ```
  public Name getName()
  ```

  Retorna un *Name* amb el nom complert (fqdn) del registre d'aquest *CameliaRecord*

- getPolicy

  ```
  public CameliaPolicy getPolicy()
  ```

  Retorna un objecte de tipus *CameliaPolicy* amb la lògica de resolució associada en aquest *CameliaRecord*

- resolveRecord

  ```
  public RRset resolveRecord(InetAddress IP, LookupService GeoIP) {
  ```

  Retorna un objecte de tipus *RRset* (tipus definit al paquet *dnsjava*) a partir de dos objectes passats per paràmetre: una adreça IP i un objecte de base de dades GeoIP; i a partir de l'objecte de tipus *CameliaPolicy* que conté l'actual *CameliaRecord*

CameliaPolicy(s), que són?

Una *CameliaPolicy* és un objecte que es crea en temps d'execució de les classes Java de *CameliaDNS* que conté la configuració d'una política de resolució per a un registre *CameliaRecord* d'una *CameliaZone*.

Durant l'execució de *CameliaDNS* pot haver tantes *CameliaPolicy* com *CameliaRecords* existeixin.

No pot haver dos *CameliaPolicy* iguals ja que per disseny l'objecte de *CameliaPolicy* conté el *Name* del *CameliaRecord* al que aplica la política, i aquest *Name* es troba guardat en format FQDN.

Cada objecte de tipus *CameliaPolicy* conté un objecte amb la política específica definida per aquest registre. No pot haver més d'una política definida per *CameliaPolicy*.

Cada objecte de tipus *CameliaRecord* conté un únic objecte de tipus *CameliaPolicy*, el qual és la representació de la configuració de la política de resolució per aquest registre. Aquest objecte de tipus *CameliaPolicy* es defineix de la següent manera (a la classe de *CameliaRecord*):

```
private CameliaPolicy policy;
```

Mètodes de CameliaPolicy

- Constructor públic CameliaPolicy

  ```
  public CameliaPolicy (Name name, int ttl, XMLSettingNode cameliaConfigXMLSettings)
  ```

  Crea un objecte del tipus *CameliaPolicy* a partir d'un *Name* que representa el nom FQDN del registre *CameliaRecord* al qual aplica aquesta política, el TTL per aquesta política i *un XMLSettingNode* que és la part del fitxer XML de la zona que conté la configuració d'aquesta política.

- getType

  ```
  public String getType()
  ```

  Retorna un *String* indicant quin es el tipus de política que conté aquest objecte *CameliaPolicy*

- isStatic

  ```
  public boolean isStatic()
  ```

  Retorna un booleà indicant si aquesta política es del tipus *CameliaStaticPolicy*

- isGeoLoc

  ```
  public boolean isGeoLoc()
  ```

  Retorna un booleà indicant si aquesta política es del tipus *CameliaTimeLocPolicy*

- isTimeLoc

  ```
  public boolean isTimeLoc()
  ```

  Retorna un booleà indicant si aquesta política es del tipus *CameliaTimeLocPolicy*

- isIPRange

  ```
  public boolean isIPRange()
  ```

  Retorna un booleà indicant si aquesta política es del tipus *CameliaIPRangePolicy*

- resolve

  ```
  public List<Record> resolve(InetAddress IP, LookupService GeoIP)
  ```

  Retorna una llista d'objectes de tipus *Record* (tipus definit al paquet *dnsjava*) a partir de dos objectes passats per paràmetre: una adreça IP i un objecte de base de dades GeoIP; i a partir de la política concreta que apliqui en aquest objecte *CameliaPolicy*

- getDefault

  ```
  public List<Record> getDefault()
  ```

  Retorna una llista d'objectes de tipus *Record* amb el valor per defecte a retornar per l'objecte *CameliaPolicy* sense passar pel motor de resolució de la política en concret.

Tipus de CameliaPolicy

Hi ha quatre tipus diferenciats de polítiques que poden ser incloses dins d'una *CameliaPolicy*, es representen amb objectes amb els següents noms a la classe principal de *CameliaPolicy*:

```
private CameliaStaticPolicy staticpolicy;
private CameliaGeoLocPolicy geolocpolicy;
private CameliaTimeLocPolicy timelocpolicy;
private CameliaIPRangePolicy iprangepolicy;
```

Aquests noms fan referència als quatre tipus de registres intel·ligents que *CameliaDNS* vol implementar.

Com s'ha comentat abans en el document per registre només pot haver una política definida, això vol dir que encara que hi hagi quatre objectes declarats només un s'inicialitzarà. Això *CameliaDNS* sap fer-ho en el constructor de *CameliaPolicy* gràcies a un petit bucle que comprova si el següent tag que ve a la configuració es un o l'altre.

Primer es defineix a nivell global de la classe de *CameliaPolicy* un petit *array* de *String* amb els noms de les polítiques. Els valors *Static*, *GeoLoc*, *TimeLoc* o *IPRange* són els tags que pot tenir la configuració d'una *CameliaPolicy*:

```
        private String[] policyType = {
                        "Test",
                        "Static",
                        "GeoLoc",
                        "TimeLoc",
                        "IPRange"
                        };
```

Després es llegeix la configuració XML intentant trobar per ordre (descartant *Test* ja que és només per proves) si sota el tag <*Policies*> existeix algun tag <*Static*>, <*GeoLoc*>, etc...

```
for (int i=1; i<policyType.length; i++) {
  ArrayList<XMLSettingNode> policy = new
    ArrayList<XMLSettingNode>(cameliaConfigXMLSettings.getSettings(policyType[i]));

  if (policy.size() > 0){
    this.type = i;

    if (this.isStatic()) {
      staticpolicy = new CameliaStaticPolicy (name, ttl, policy.get(0));
    }
    else if (this.isGeoLoc()) {
      geolocpolicy = new CameliaGeoLocPolicy (name, ttl, policy.get(0));
    }
    else if (this.isTimeLoc()) {
      timelocpolicy = new CameliaTimeLocPolicy (name, ttl, policy.get(0));
    }
    else if (this.isIPRange()) {
      iprangepolicy = new CameliaIPRangePolicy (name, ttl, policy.get(0));
    }
  }
}
```

Objectes CameliaStaticPolicy

Un *CameliaStaticPolicy* és un objecte que s'inicialitza dins de l'objecte *CameliaPolicy* per un *CameliaRecord* dins d'una *CameliaZone* durant l'execució de *CameliaDNS*.

CameliaStaticPolicy és una de les quatre possibles polítiques que pot contenir un objecte de tipus *CameliaPolicy*.

Durant l'execució de *CameliaDNS* pot haver tantes *CameliaStaticPolicy* com *CameliaRecords* existeixin.

Cada objecte de tipus *CameliaStaticPolicy* conté dos objectes principalment:

- Llistat de *Records* llegits de la configuració
- *Record* per defecte (el primer de la configuració)

Aquests objectes es defineixen a l'inici de la classe *CameliaStaticPolicy* de la següent manera:

```
private ArrayList<Record> recordList;
private Record defaultRecord;
```

Mètodes de CameliaStaticPolicy

- Constructor públic CameliaStaticPolicy

  ```
  public CameliaStaticPolicy
    (Name name, int ttl, XMLSettingNode cameliaConfigXMLSettings)
  ```

  Crea un objecte del tipus *CameliaStaticPolicy* a partir d'un *Name* que representa el nom FQDN del registre *CameliaRecord* al qual aplica aquesta política, el TTL per aquesta política i un *XMLSettingNode* que és la part del fitxer XML de la zona que conté la configuració de la política *Static*.

- getRecords

  ```
  public ArrayList<Record> getRecords()
  ```

  Retorna un *ArrayList* de *Record* amb la llista de registres inclosos en aquesta política (l'objecte *recordList*)

- getDefault

  ```
  public ArrayList<Record> getDefault()
  ```

  Retorna el *Record* per defecte (l'objecte *defaultRecord*) per aquesta política

Objectes CameliaTimeLocPolicy

Un *CameliaTimeLocPolicy* és un objecte que s'inicialitza dins de l'objecte *CameliaPolicy* per un *CameliaRecord* dins d'una *CameliaZone* durant l'execució de *CameliaDNS*.

CameliaTimeLocPolicy és una de les quatre possibles polítiques que pot contenir un objecte de tipus *CameliaPolicy*.

Durant l'execució de *CameliaDNS* pot haver tantes *CameliaTimeLocPolicy* com *CameliaRecords* existeixin.

Cada objecte de tipus *CameliaTimeLocPolicy* conté dos objectes principalment:

- Llistat de *Slice* llegits de la configuració
- *Record* per defecte definit a la configuració

Aquests objectes es defineixen a l'inici de la classe *CameliaTimeLocPolicy* de la següent manera:

```
private Record defaultRecord;
private ArrayList<CameliaTimeLocSlice> slices;
```

Mètodes de CameliaTimeLocPolicy

- Constructor públic CameliaTimeLocPolicy

  ```
  public CameliaTimeLocPolicy
    (Name name, int ttl, XMLSettingNode cameliaConfigXMLSettings)
  ```

  Crea un objecte del tipus *CameliaTimeLocPolicy* a partir d'un *Name* que representa el nom FQDN del registre *CameliaRecord* al qual aplica aquesta política, el TTL per aquesta política i un *XMLSettingNode* que és la part del fitxer XML de la zona que conté la configuració de la política *TimeLoc*.

- getRecords

  ```
  public ArrayList<Record> getRecords()
  ```

  Retorna un *ArrayList* de *Record* amb la llista de registres a respondre per aquesta petició després d'haver avaluat l'hora d'entrada d'aquesta respecte a la llista de *Slices* configurats

- getDefault

  ```
  public ArrayList<Record> getDefault()
  ```

  Retorna el *Record* per defecte (l'objecte *defaultRecord*) per aquesta política

Objectes CameliaGeoLocPolicy

Un *CameliaGeoLocPolicy* és un objecte que s'inicialitza dins de l'objecte *CameliaPolicy* per un *CameliaRecord* dins d'una *CameliaZone* durant l'execució de *CameliaDNS*.

CameliaGeoLocPolicy és una de les quatre possibles polítiques que pot contenir un objecte de tipus *CameliaPolicy*.

Durant l'execució de *CameliaDNS* pot haver tantes *CameliaGeoLocPolicy* com *CameliaRecords* existeixin.

Cada objecte de tipus *CameliaGeoLocPolicy* conté dos objectes principalment:

- *HashMap* de *CountryCode* i valors a entregar llegits de la configuració
- *Record* per defecte definit a la configuració

Aquests objectes es defineixen a l'inici de la classe *CameliaGeoLocPolicy* de la següent manera:

```
private Record defaultRecord;
private HashMap<String, ArrayList<Record>> countryRecordMap;
```

Mètodes de CameliaGeoLocPolicy

- Constructor públic CameliaGeoLocPolicy

  ```
  public CameliaGeoLocPolicy
    (Name name, int ttl, XMLSettingNode cameliaConfigXMLSettings)
  ```

  Crea un objecte del tipus *CameliaGeoLocPolicy* a partir d'un *Name* que representa el nom FQDN del registre *CameliaRecord* al qual aplica aquesta política, el TTL per aquesta política i un *XMLSettingNode* que és la part del fitxer XML de la zona que conté la configuració de la política *GeoLoc*.

- getRecords

  ```
  public ArrayList<Record> getRecords(InetAddress IP, LookupService GeoIP)
  ```

  Retorna un *ArrayList* de *Record* amb la llista de registres a respondre per aquesta petició després d'haver avaluat la IP originaria de la petició amb la base de dades de localitzacions d'IPs

- getDefault

  ```
  public ArrayList<Record> getDefault()
  ```

  Retorna el *Record* per defecte (l'objecte *defaultRecord*) per aquesta política

- mètode privat checkGeoIP

  ```
  private ArrayList<Record> checkGeoIP (InetAddress IP, LookupService GeoIP)
  ```

  Aquest mètode es crida des de *getRecords.* Ofereix la mateixa funcionalitat que *getRecords* però des d'un punt de vista privat i reparteix la feina amb un altre mètode. Comprova la IP contra la base de dades d'IPs per obtenir el país d'aquesta, i crida a un altre mètode que és el que comprova si per aquest país existeix una entrada al *HashMap countryRecordMap*

- mètode privat checkCountryRecords

  ```
  private ArrayList<Record> checkCountryRecords(String country)
  ```

  Aquest mètode es crida des de *checkGeoIP*. Retorna un *ArrayList* de *Record* amb la llista de registres a respondre per aquest *CameliaRecord* per al país *country*. El *String country* es comprovat contra el *HashMap countryRecordMap* per obtenir la llista.

Objectes CameliaIPRangePolicy

Un *CameliaIPRangePolicy* és un objecte que s'inicialitza dins de l'objecte *CameliaPolicy* per un *CameliaRecord* dins d'una *CameliaZone* durant l'execució de *CameliaDNS*.

CameliaIPRangePolicy és una de les quatre possibles polítiques que pot contenir un objecte de tipus *CameliaPolicy*.

Durant l'execució de *CameliaDNS* pot haver tantes *CameliaIPRangePolicy* com *CameliaRecords* existeixin.

Cada objecte de tipus *CameliaIPRangePolicy* conté dos objectes principalment:

- Llistat de *Subnet* llegits de la configuració
- *Record* per defecte definit a la configuració

Aquests objectes es defineixen a l'inici de la classe *CameliaIPRangePolicy* de la següent manera:

```
private Record defaultRecord;
private ArrayList<CameliaIPRangeSubnet> subnets;
```

Mètodes de CameliaIPRangePolicy

- Constructor públic CameliaIPRangePolicy

  ```
  public CameliaIPRangePolicy
    (Name name, int ttl, XMLSettingNode cameliaConfigXMLSettings)
  ```

  Crea un objecte del tipus *CameliaIPRangePolicy* a partir d'un *Name* que representa el nom FQDN del registre *CameliaRecord* al qual aplica aquesta política, el TTL per aquesta política i un *XMLSettingNode* que és la part del fitxer XML de la zona que conté la configuració de la política *IPRange*.

- getRecords

  ```
  public ArrayList<Record> getRecords(InetAddress IP)
  ```

  Retorna un *ArrayList* de *Record* amb la llista de registres a respondre per aquesta petició després d'haver avaluat la IP originaria de la petició respecte la llista de *Subnet* configurats

- getDefault

  ```
  public ArrayList<Record> getDefault()
  ```

  Retorna el *Record* per defecte (l'objecte *defaultRecord*) per aquesta política

Objectes CameliaTimeLocSlice

Un *CameliaTimeLocSlice* és un objecte que s'inicialitza dins de l'objecte *CameliaTimeLocPolicy* per una *CameliaPolicy* dins d'un *CameliaRecord*.

CameliaTimeLocSlice és usat per *CameliaTimeLocPolicy* per representar una *Slice* de les definides al fitxer de configuració.

Cada objecte de tipus *CameliaTimeLocSlice* conté tres objectes principalment:

- Llistat de *Record* llegits de la configuració
- Temps d'inici de la franja de temps
- Temps d'acabament de la franja de temps

Aquests objectes es defineixen a l'inici de la classe *CameliaTimeLocSlice* de la següent manera:

```
private ArrayList<Record> records;
private String timeIni;
private String timeEnd;
```

Mètodes de CameliaTimeLocSlice

- Constructor públic CameliaTimeLocSlice

  ```
  public CameliaTimeLocSlice
    (ArrayList<Record> records, String timeIni, String timeEnd)
  ```

  Crea un objecte del tipus *CameliaTimeLocSlice* a partir d'una llista de *Record* proporcionada, l'hora d'inici de la franja horària i l'hora d'acabament en format *String* els dos. Aquest mètode es cridat des del constructor de la classe *CameliaTimeLocPolicy* quan s'estan carregant els diferents *Slice* per la política de franja horària.

- getTimeIni

  ```
  public String getTimeIni()
  ```

  Retorna un *String* amb el temps d'inici de la franja horària d'aquest *Slice*

- getTimeEnd

  ```
  public String getTimeEnd()
  ```

  Retorna un *String* amb el temps de finalització de la franja horària d'aquest *Slice*

- getRecords

  ```
  public ArrayList<Record> getRecords()
  ```

  Retorna el llistat de *Record* associat al *Slice* actual

Objectes CameliaIPRangeSubnet

Un *CameliaIPRangeSubnet y* és un objecte que s'inicialitza dins de l'objecte *CameliaIPRangePolicy* per una *CameliaPolicy* dins d'un *CameliaRecord*.

CameliaIPRangeSubnet és usat per *CameliaIPRangePolicy* per representar una *Subnet* de les definides al fitxer de configuració.

Cada objecte de tipus *CameliaIPRangeSubnet* conté quatre objectes principalment:

- Llistat de *Record* llegits de la configuració
- Subxarxa amb la que fer match
- Prefix de la xarxa
- Màscara de la xarxa

Aquests objectes es defineixen a l'inici de la classe *CameliaIPRangeSubnet* de la següent manera:

```
private ArrayList<Record> records;
private String subnet;
private String networkPrefix;
private int subnetMask;
```

Mètodes de CameliaIPRangeSubnet

- Constructor públic CameliaIPRangeSubnet

  ```
  public CameliaIPRangeSubnet (ArrayList<Record> records, String subnet)
  ```

  Crea un objecte del tipus *CameliaIPRangeSubnet* a partir d'una llista de *Record* proporcionada, i la xarxa en format *String*. Aquest mètode es cridat des del constructor de la classe *CameliaIPRangePolicy* quan s'estan carregant les diferents *Subnet* per la política de subnetting.

  A més a més, el constructor també assigna a unes variables globals a nivell de classe els valors de xarxa, prefix, i màscara per tenir-los més accessible per treballar amb ells des d'altres mètodes.

  ```
  String[] subnetParts = subnet.split("/");
  this.networkPrefix = subnetParts[0];
  this.subnetMask = Integer.parseInt(subnetParts[1]);
  ```

- getSubnet

  ```
  public String getSubnet()
  ```

  Retorna un *String* amb la definició de la xarxa associada a aquest objecte

- getNetworkPreffix

  ```
  public String getNetworkPrefix()
  ```

  Retorna un *String* amb el prefix de xarxa, sense la màscara

- getSubnetMask

  ```
  public int getSubnetMask()
  ```

  Retorna en format *int* la màscara d'aquesta Subnet

- getRecords

  ```
  public ArrayList<Record> getRecords()
  ```

  Retorna el llistat de *Record* associat a la *Subnet* actual

Funcionalitats afegides amb Camelia DNS

Registres estàtics

Els **registres estàtics** són els que internament *CameliaDNS* coneix com *CameliaRecord* i que tenen un objecte de tipus ***CameliaStaticPolicy*** inicialitzat dins del seu objecte *CameliaPolicy*.

Els objectes de tipus *CameliaStaticPolicy* són el que contenen la informació relativa a resoldre un registre *CameliaRecord* de manera estàtica.

La configuració d'un registre de Camelia amb *StaticPolicy* llueix així:

```
<Camelia>
        <Record>
            <Name>mx3</Name>
            <RecordTTL>90</RecordTTL>
            <Policies>
                <Static>
                    <Value>72.28.11.3</Value>
                </Static>
            </Policies>
        </Record>
 </Camelia>
```

Sota el tag *Static* es defineixen tants tags *Value* com es desitgin.

Aquests registres en funcionalitat són molt semblants al que ens aporta un DNS clàssic:

- Múltiples entrades A pel mateix *CameliaRecord*: es retorna un pool d'IPs com a resposta. Aquestes IPs en cada resposta vindran ordenades d'una manera diferent de cara a fer un balanceig round-robin de les IPs.

- Múltiples entrades CNAME pel mateix *CameliaRecord*: el RFC del DNS no permet retornar un pool de registres de CNAME però el que si que pot fer *CameliaDNS* es que donat un llistat de N noms retorni un però no sempre el mateix. Es a dir podem fer round-robin de registres CNAME. Exemple de configuració:

```
<Policies>
    <Static>
        <Value>host1.foo.com.</Value>
        <Value>host2.foo.com.</Value>
    </Static>
</Policies>
```

Registres dinàmics

Els registres dinàmics són tots aquells que per tal d'entregar la resposta avaluen condicions circumstancials com la IP procedent de la petició o el moment en el que ha arribat la petició per escollir si es retorna un resultat o un altre.

Resolució per franja horària

Els registres **dinàmics per franja horària** són els que internament *CameliaDNS* coneix com *CameliaRecord* i que tenen un objecte de tipus ***CameliaTimeLocPolicy*** inicialitzat dins del seu objecte *CameliaPolicy*.

Els objectes de tipus *CameliaTimeLocPolicy* són el que contenen la informació relativa a resoldre un registre *CameliaRecord* en funció de l'hora del dia en la que ens trobem.

La configuració d'un registre de Camelia amb *TimeLocPolicy* llueix així:

```
<Camelia>
        <Record>
            <Name>admin</Name>
            <RecordTTL>300</RecordTTL>
            <Policies>
                <TimeLoc>
                    <Slice>
                        <TimeIni>00:00:00</TimeIni>
                        <TimeEnd>08:00:00</TimeEnd>
                        <Value>192.168.5.1</Value>
                    </Slice>
                    <Slice>
                        <TimeIni>08:00:00</TimeIni>
                        <TimeEnd>13:15:00</TimeEnd>
                        <Value>192.168.5.2</Value>
                        <Value>192.168.6.2</Value>
                    </Slice>
                    <Slice>
                        <TimeIni>13:15:00</TimeIni>
                        <TimeEnd>23:59:59</TimeEnd>
                        <Value>192.168.5.3</Value>
                        <Value>192.168.6.3</Value>
                        <Value>192.168.7.3</Value>
                    </Slice>
                    <DefaultValue>192.168.5.222</DefaultValue>
                </TimeLoc>
            </Policies>
        </Record>
 </Camelia>
```

Sota el tag *TimeLoc* es defineixen tants *Slice* com es desitgin. Sempre procurant abastar les 24h del dia, i que no hi hagi solapaments.

Si els *Slice* definits no abasten totes les hores quan entri una petició a una hora no compresa en cap *Slice* es retornarà el *DefaultValue*.

Si hi ha un solapament, tindrà preferència el primer en que l'hora actual de la petició faci match (el primer per ordre de lectura de la configuració).

Resolució per GEO localització

Els registres **dinàmics per GEO localització** són els que internament *CameliaDNS* coneix com *CameliaRecord* i que tenen un objecte de tipus ***CameliaGeoLocPolicy*** inicialitzat dins del seu objecte *CameliaPolicy*.

Els objectes de tipus *CameliaGeoLocPolicy* són el que contenen la informació relativa a resoldre un registre *CameliaRecord* en funció de la localització geogràfica de la IP que ha enviat la petició.

La resolució per GEO Localització per funcionar necessita un fitxer que contingui la base de dades d'IPs i xarxes mundials que pertanyen a cada país. *CameliaDNS* funciona amb la versió *GeoLite Country* de Maxmind que es pot descarregar des d'aquí: http://dev.maxmind.com/geoip/legacy/geolite/. El fitxer *GeoIP.dat* s'ha d'ubicar al directory *geoip* dins del directori arrel d'*EagleDNS*.

La configuració d'un registre de Camelia amb *GeoLocPolicy* llueix així:

```
<Camelia>
        <Record>
            <Name>mx3</Name>
            <RecordTTL>90</RecordTTL>
            <Policies>
                <GeoLoc>
                    <Country>
                        <CountryName>Ireland</CountryName>
                        <CountryCode>IE</CountryCode>
                        <Value>54.14.57.234</Value>
                        <Value>54.14.57.235</Value>
                        <Value>54.14.58.120</Value>
                        <Value>54.14.58.121</Value>
                    </Country>
                    <DefaultValue>87.15.47.130</DefaultValue>
                </GeoLoc>
            </Policies>
        </Record>
 </Camelia>
```

Sota el tag *GeoLoc* es defineixen tants *Country* com es desitgin.

Dins de cada *Country* s'ha de definir el *CountryName* i *CountryCode*, sempre seguint la nomenclatura de Maxmind, el proveïdor del fitxer de base de dades d'IPs.

No es necessari definir tots el països però si un valor per defecte. Ja que en el cas de que una petició vingui d'un país no contemplat s'ha de retornar algun valor.

No es poden definir més d'un país per *Country*. En el cas de voler definir tres països amb la mateixa configuració s'han de crear tres entrades *Country* amb els seus corresponents *CountryName*, *CountryCode* i *Values*.

Resolució per subnetting

Els registres **dinàmics per subnetting** són els que internament *CameliaDNS* coneix com *CameliaRecord* i que tenen un objecte de tipus ***CameliaIPRangePolicy*** inicialitzat dins del seu objecte *CameliaPolicy*.

Els objectes de tipus *CameliaIPRangePolicy* són el que contenen la informació relativa a resoldre un registre *CameliaRecord* en funció de a quina xarxa que es defineixi en la configuració pertany la IP originària de la petició.

La configuració d'un registre de Camelia amb *IPRangePolicy* llueix així:

```
<Camelia>
        <Record>
            <Name>backoffice</Name>
            <RecordTTL>600</RecordTTL>
            <Policies>
                <IPRange>
                    <Subnet>
                        <Match>227.0.99.23/32</Match>
                        <Value>92.143.77.17</Value>
                    </Subnet>
                    <Subnet>
                        <Match>52.18.75.69/32</Match>
                        <Value>34.226.156.27</Value>
                    </Subnet>
                    <Subnet>
                        <Match>10.15.0.0/16</Match>
                        <Value>10.15.77.4</Value>
                    </Subnet>
                    <Subnet>
                        <Match>10.15.3.0/24</Match>
                        <Value>10.15.3.3</Value>
                    </Subnet>
                    <DefaultValue>57.62.185.99</DefaultValue>
                </IPRange>
            </Policies>
        </Record>
 </Camelia>
```

Sota el tag *IPRange* es defineixen tantes *Subnet* com es desitgin.

Dins de cada *Subnet* s'ha de definir el *Match*, que és la xarxa a la qual una IP pot pertànyer. El format acceptat es el CIDR (Classless Inter-Domain Routing), basat en posar el prefix de la xarxa seguit d'una barra amb el nombre de bits de la màscara.

No es necessari definir totes les xarxes possibles però si un valor per defecte. Ja que en el cas de que una petició vingui d'una xarxa no contemplada s'ha de retornar algun valor.

No es pot definir més d'un *Match* per *Subnet*.

Per un *CameliaRecord* que tingui la política *CameliaIPRangePolicy* s'examinaran sempre totes les *Subnet* per tal de retornar el valor més acurat possible. Si la IP originària de la petició està inclosa en varies *Subnet* es retornarà el *Value* de la xarxa més petita (amb un nombre de bits de màscara més alt), tal i com es fa en networking a les taules de rutes.

Infraestructura

Per tal de dur a terme aquest PFC s'ha necessitat una petita infraestructura per donar suport al desenvolupament del software, desplegament de canvis i proves del servei DNS.

Subversion + WebSVN

El primer de tot va ser buscar un sistema que permetés portar un control de revisions dels canvis realitzats sobre el codi.

La experiència en aquell moment es basava en SVN principalment així que es va decidir tirar endavant amb aquesta solució. Al 2010 es van muntar un parell de servidors SVN dedicats al projecte.

Hi havia un que era d'us compartit entre els membres de l'equip i un altre d'ús local on es van anar pujant inicialment les primeres revisions de CameliaDNS.

Amb l'aturada del projecte el servidor local és el que va quedar amb la còpia més actualitzada del codi, es va fer un backup (*svndump* per guardar totes les revisions) i el fitxer es va guardar en un lloc segur.

Anys més tard amb la represa del PFC vaig haver de fer arqueologia entre els meus backups per trobar el *dump* de *CameliaDNS*, i vaig pensar que en aquesta ocasió seria millor tenir el servidor de SVN en un proveïdor de Cloud. Com per motius professionals habitualment treballo amb Amazon AWS vaig decidir obrir un compte gratuït per allotjar el servei de SVN.

En aquesta ocasió vaig decidir afegir també la peça de WebSVN per poder facilitar-me la vida a l'hora de visualitzar els canvis de revisions, o fer comparatives entre elles. Es pot fer servir el CLI però en certs moments i per canvis molt grans de codi es preferible tenir una GUI al davant.

El sistema operatiu seleccionat per la instancia d'Amazon va ser Ubuntu. Hi ha actualització sovint de paquets importants de sistema, es fàcil trobar paquets ja preparats als repositoris, i a més estic bastant familiaritzat perquè els últims anys és una de les distribucions Linux amb les que més he treballat.

En el que respecte al "hardware" de la instancia, tant aquest com d'altres que s'han fet servir sempre han estat les més petites ja que Amazon les ofereix gratuïtament durant un any. No hi havia necessitat de pagar quan amb les especificacions de les màquines petites és suficient.

Instal·lació de Subversion + WebSVN

La instal·lació es molt fàcil, es suficient amb descarregar els paquets i dependències amb *apt*. El gestor de paquets fa la resta i ho instal·la.

```
# apt-get install subversion apache2 libssl-dev libcurl4-openssl-dev websvn php5-cli
    php5-common libapache2-mod-php5 libapache2-mod-svn
```

Configuració de Subversion + WebSVN

A continuació s'han de configurar el serveis. S'ha de tenir en compte que s'ha de configurar per una banda Subversion+Apache, i WebSVN+Apache.

- Creem el directori on ubicarem els repositoris: */opt/subversion*
- I el seus subdirectoris per configuracions i dades
  - */opt/subversion/conf*
  - */opt/subversion/repos*
- Definim permisos per accedir als repositoris:

```
# cat /opt/subversion/conf/svn-policy-file
[camelia:/]
hugo.peris = rw

# cat /opt/subversion/conf/svn-auth-file
hugo.peris:<hashmd5>
```

- Importem el backup

```
# svnadmin create /opt/subversion/repos/camelia
# svnadmin load /opt/subversion/repos/camelia < cameliabackup.dump
```

- Configurem Apache per accedir a Subversion

```
# cat /etc/apache2/sites-available/000-default.conf

<Location /svn>
   DAV svn
   SVNParentPath /opt/subversion/repos
   AuthzSVNAccessFile /opt/subversion/conf/svn-policy-file
   AuthName "SVN Repositories"
   AuthType Basic
   AuthUserFile /opt/subversion/conf/svn-auth-file
   Require valid-user
</Location>
```

- Creem un fitxer de configuració d'Apache bàsic per WebSVN

```
# cat /etc/apache2/conf-available/websvn

Alias /websvn /usr/share/websvn

<Directory /usr/share/websvn>
  AuthType Basic
  AuthName WEBsvn
  AuthUserFile /opt/subversion/conf/svn-auth-file
  Require valid-user
  Order deny,allow
  Deny from all
  Satisfy any
</Directory>
```

- I l'activem

```
# ln -s /etc/apache2/conf-available/websvn /etc/apache2/conf-enabled/websvn.conf
```

Puppet

Després d'un temps d'haver reprès el PFC vaig pensar que si volia replicar la màquina de SVN en una altre igual per fer alta disponibilitat o potser per recuperar-la si la perdia no tenia backup de les configuracions de sistema.

Encara que fossin petites coses, quan es comencessin a sumar petites configuracions per un lloc i per un altre faria un total de temps molt gran si volia desplegar una altra màquina igual.

A més vaig pensar que a part de desenvolupar en local i provar-ho en el meu portàtil, estaria bé pujar el codi a una plataforma que simulés ser un entorn de proves + producció.

Donats aquests pensaments, era d'esperar que voldria desplegar més màquines (recordem: instàncies d'Amazon). Aprofitant la meva experiència en *Puppet* (és el meu dia a dia laboral) vaig decidir instal·lar un *Puppet master* per aglutinar totes les configuracions de sistema i així aprofitar-me d'aquesta eina per poder desplegar canvis genèrics en les N-màquines que tingués.

Fins ara tenia una única màquina, la de SVN (a partir d'ara la anomenarem **vcs1**). I pretenia instal·lar una nova per *Puppet*, batejada com **adm1**. Però també en previsió de desplegar *CameliaDNS* vaig pensar en desplegar dos màquines més una simulant un entorn de Staging i una altre simulant producció: **stg1** i **dns1** es van dir.

El que primer faria es configurar en els manifestos de *Puppet* la actual màquina **vcs1**, i després la pròpia màquina de *Puppet* la "puppetitzaria" també. A més d'això la configuració de *Puppet* la afegiria com un projecte nou al Subversion i així podria portar un registre del canvis a nivell de configuracions de sistema i serveis.

Instal·lació de Puppet

Puppet està programat en *Ruby* i per que els clients accedeixin al master es pot fer fent-lo córrer de forma standalone. O com es fa de manera més habitual a través d'*Apache*.

Apache exposa el port 8140 a través del qual amb el mòdul *passenger* envia les peticions a l'intèrpret de *Ruby* que fa córrer *Puppet*.

Seguint la documentació oficial de *Puppetlabs* instal·lem tot el software necessari.

```
# apt-get puppet install apache2 ruby1.8-dev rubygems
# a2enmod ssl
# a2enmod headers
# gem install rack passenger
# passenger-install-apache2-module
```

Configuració de Puppet

- Abans de res s'han de crear els directoris necessaris pel *rack*
  - */etc/puppet/rack*
  - */etc/puppet/rack/tmp*
  - */etc/puppet/rack/public*

- Després s'ha copiar el fitxer *config.ru* que es pot trobar al *GitHub* de *Puppetlabs* dins del directori del *rack*
  - */etc/puppet/rack/config.ru*

- El següent pas es habilitar el *VirtualHost* perquè l'*Apache* pugui servir *Puppet*

```
# cat /etc/apache2/sites-available/puppetmaster.conf
** (versió simplificada)

<IfModule mod_passenger.c>
  PassengerRoot /var/lib/gems/1.9.1/gems/passenger-5.0.15
  PassengerDefaultRuby /usr/bin/ruby1.9.1
  PassengerAppRoot /etc/puppet/rack
</IfModule>

Listen 8140
<VirtualHost *:8140>
    SSLEngine On
    DocumentRoot /etc/puppet/rack/public/

    <Directory /etc/puppet/rack/>
      Options None
      AllowOverride None
      # Apply the right behavior depending on Apache version.
      <IfVersion < 2.4>
        Order allow,deny
        Allow from all
      </IfVersion>
      <IfVersion >= 2.4>
        Require all granted
      </IfVersion>
    </Directory>

</VirtualHost>
```

Usant Puppet

Definint els nodes

A partir d'aquí ja es pot fer servir *Puppet* lliurement. Una de les primeres coses a fer es definir un manifest on es declararan les màquines, aquest manifest habitualment se li diu *nodes.pp*

Per la infraestructura d'aquest PFC és molt simple

```
# cat /etc/puppet/manifests/nodes.pp

node default {

  class { 'system::base': }
  class { 'admin::users': }

  case $::role{

    vcs: {
      # vcs nodes
      class { 'subversion::server': }
      class { 'backup::svn': }
    }

    stg: {
      # stg nodes
      class { 'camelia::cameliadns': environment => 'test' }
    }

    adm: {
      # adm nodes
      class { 'subversion::client': }
      class { 'master::camelia': }
      class { 'puppet::master': }
    }

    dns: {
      # dns nodes
      class { 'camelia::cameliadns': environment => 'prod' }
    }
  }
}
```

Tots les màquines (a *Puppet* s'anomenen nodes) les passem pel node *default* i dins d'aquest llegim la variable global *role*, o com es diu en idioma de *Puppet*: *fact*. Puppet fa servir un software de recolzament per extreure informació dels sistemes on corre, aquest software es diu *Facter*.

Facter extreu una llista de *facts* com per exemple el sistema operatiu o versió de kernel, però també es poden introduir *facts* personalitzats. En aquest cas s'ha introduït un *fact* personalitzat que ens diu el *role* de la màquina. Els *facts* es programen en *Ruby* i es poden copiar manualment a la màquina desitjada o es pot fer que el propi *Puppet* els distribueixi. En aquest *Puppet* el *fact* està desplegat de manera automàtica.

La manera que té el *fact* de saber si una màquina és d'un *role*, o un altre, i per tant *Puppet* després aplicar unes configuracions o unes altres es la següent:

```
# cat /etc/puppet/modules/custom/lib/facter/nodes.rb
require 'facter'

Facter.add("role") do
  node_role = Facter.value(:ec2_security_groups).sub(/^(.*,)?(\w+)(,.*)?$/, '\2')
  setcode do
    node_role
  end
end
```

Els security groups de la instancia d'Amazon marquen la diferència. En aquest cas el que s'ha fet és que cada màquina s'ha configurat amb un security group amb un nom concret:

- vcs
- adm
- stg
- dns

Gràcies a això *Puppet* sap si aplicar unes configuracions o unes altres segons la màquina. I quan es vulguin afegir màquines noves d'un tipus només ens hem de preocupar de configurar-les amb el security group corresponent.

Manifestos per vcs1

La idea inicial era tenir definit a *Puppet* la màquina **vcs1** i començarem per aquí abans de res. Als manifestos declarem alguns fitxers de configuració i paquets que volem que estiguin instal·lats.

A la màquina **vcs1** principalment tenim corrent Subversion + Websvn. Així que existeix un manifest que conté les configuracions necessàries per desplegar aquests serveis a la màquina.

- Fitxer ubicat a **adm1**:*/etc/puppet/modules/subversion/manifests/server.pp*

Però tot i així encara no estem salvats davant d'una catàstrofe així que existeix un mòdul de backup.

Aquest mòdul de backup s'ocupa de desplegar un script i configuracions que s'encarreguen de fer un *svndump* dels repositoris importants:

- camelia (el software de CameliaDNS)
- camelia-puppet (les configuracions de Puppet)

I de pujar-los a un altre proveïdor de Cloud, s'ha escollit Mega ja que ofereix un bon nivell d'encriptació i 50GB d'espai de franc.

Per pujar els fitxers a Mega fan falta unes utilitats anomenades Megatools que s'instal·len via Puppet també des de el mòdul de backup:

```
class backup::svn {

  require ::megatools::install
```

El mòdul de backup de svn també desplega un script i un cron per realitzar el backup

```
  file { '/opt/backup/svn-backup.sh':
    ensure  => present,
    owner   => 'root',
    group   => 'root',
    mode    => 755,
    source  => 'puppet:///modules/backup/svn/svn-backup.sh',
    require => File['/opt/backup'],
  }

  cron { 'svn-backup.sh' :
    command => '/opt/backup/svn-backup.sh > /opt/backup/logs/svn-backup.`date
+\%s`.log 2>&1',
    user    => 'root',
    minute  => '7',
    hour    => '11',
    require => File['/opt/backup/svn-backup.sh'],
  }
```

Manifestos per adm1

La màquina **adm1** primordialment serveix *Puppet*, així que ens interessa tenir controlats els fitxers de *Puppet* amb la pròpia eina.

Això ho fa el manifest de */etc/puppet/modules/puppet/master.pp*

```
# cat /etc/puppet/modules/puppet/manifests/master.pp
** (versió simplificada)

class puppet::master {

  ## puppetmaster conf
  file { [ '/etc/puppet',
           '/etc/puppet/files',
           '/etc/puppet/rack',
           '/etc/puppet/rack/tmp',
           '/etc/puppet/rack/public', ]: }
  file { '/etc/puppet/auth.conf': }
  file { '/etc/puppet/autosign.conf': }
  file { '/etc/puppet/fileserver.conf': }
  file { '/etc/puppet/puppet.conf': }
  file { '/etc/puppet/rack/config.ru': }
  file { '/etc/puppet/etckeeper-commit-post': }
  file { '/etc/puppet/etckeeper-commit-pre': }
  ## puppetmaster certificates
  file { '/var/lib/puppet/ssl/certs/puppetmaster.pem': }
  file { '/var/lib/puppet/ssl/private_keys/puppetmaster.pem': }
  file { '/var/lib/puppet/ssl/ca/ca_crt.pem': }
  file { '/var/lib/puppet/ssl/ca/ca_crl.pem': }
  ## apache conf
  file { '/etc/apache2/sites-available/puppetmaster.conf': }
}
```

Manifestos per stg1 i dns1

Aquestes màquines a efectes pràctics són iguals amb l'única diferència que a les de tipus *stg* es desplega una versió del software de *CameliaDNS* de proves, i a les de tipus *dns* es desplega la de producció.

Tot i que Puppet no està pensat per fer desplegaments d'aquest tipus de software vaig pensar que podia aprofitar que el muntava per establir un master de versions de codi proves/producció a la màquina **adm1** i desplegar els software via Puppet.

Primer de tot el manifest *cameliadns.pp* defineix l'usuari de CameliaDNS, el servei, i crida al deploy del propi software.

```
# cat modules/camelia/manifests/cameliadns.pp
** (versió simplificada)

class camelia::cameliadns ( $environment = 'test' ) {

  require ::camelia::user

  class { 'camelia::deploy': environment => $environment }

  service { 'camelia':
    ensure  => running,
    enable  => true,
  }
}
```

Llavors al manifest *deploy.pp* és on es defineix que es vol copiar el contigut del master, aquest contigut es troba al directori d'entorn corresponent i dependrà de si estem en una màquina de tipus *stg* o *dns.*

```
# cat modules/camelia/manifests/deploy.pp

class camelia::deploy ( $environment = 'test' ) {

  require ::camelia::user
  require ::camelia::java

  file { '/opt/camelia/cameliadns' :
    ensure  => directory,
    owner   => 'camelia',
    group   => 'camelia',
    recurse => true,
    source  => "puppet:///modules/camelia/environment/${environment}/",
    require => [ File['/opt/camelia'], User['camelia'], ],
  }
}
```

Tecnologies usades

Java

Tecnologia per excel·lència del projecte. Ja que el software de *CameliaDNS* es bassa en Java.

Tenint en compte el moment en el que es va iniciar el projecte, 2010, i la posterior represa d'aquest, la versió feta servir no és la més actualitzada. Va haver un intent de compilar en una versió actual (1.8 i 1.7) però no va funcionar. No era tan sols les classes de *CameliaDNS* sinó també les *d'EagleDNS* i les llibreries que feien servir els dos mòduls.

Estava tot pensat per funcionar en Java 1.5 i es va seguir així ja que actualitzar el codi gairebé línia per línia implicava tornar a començar de cero.

Tot i això s'ha de dir que si bé *EagleDNS* es va haver de quedar en la versió 1.5, *CameliaDNS* es va poder pujar de versió de Java fins la 1.6 update 65.

Programant *CameliaDNS* s'han hagut d'incloure i compilar mòduls extra. Un d'ells és per exemple el de *StandardUtils* de *unlogic.se*.

També els mòdul de *log4j* i *GeoIP* de Maxmind, que conté el tipus *LookupService* utilitzat per *CameliaDNS*.

Degut a que es pretenia integrar amb Cassandra ha quedat en el codi alguna classe Java que fa referència a això. Podria ser un punt de partida per evolucionar cap aquí, o es podria esborrar, a efectes funcionals no afecta al servei tal i com està ara mateix.

Encara que el software està compilat en 1.5 i 1.6. A les màquines de proves i producció aquest pot córrer perfectament sota 1.8 update 60 que és l'última versió disponible.

Subversion

La segona peça important del projecte. Gràcies a haver fet servir SVN des del principi he estat capaç de recuperar l'històric de la feina feta anys enrere i seguir les meves pròpies passes fins l'estat del codi en el moment de la represa.

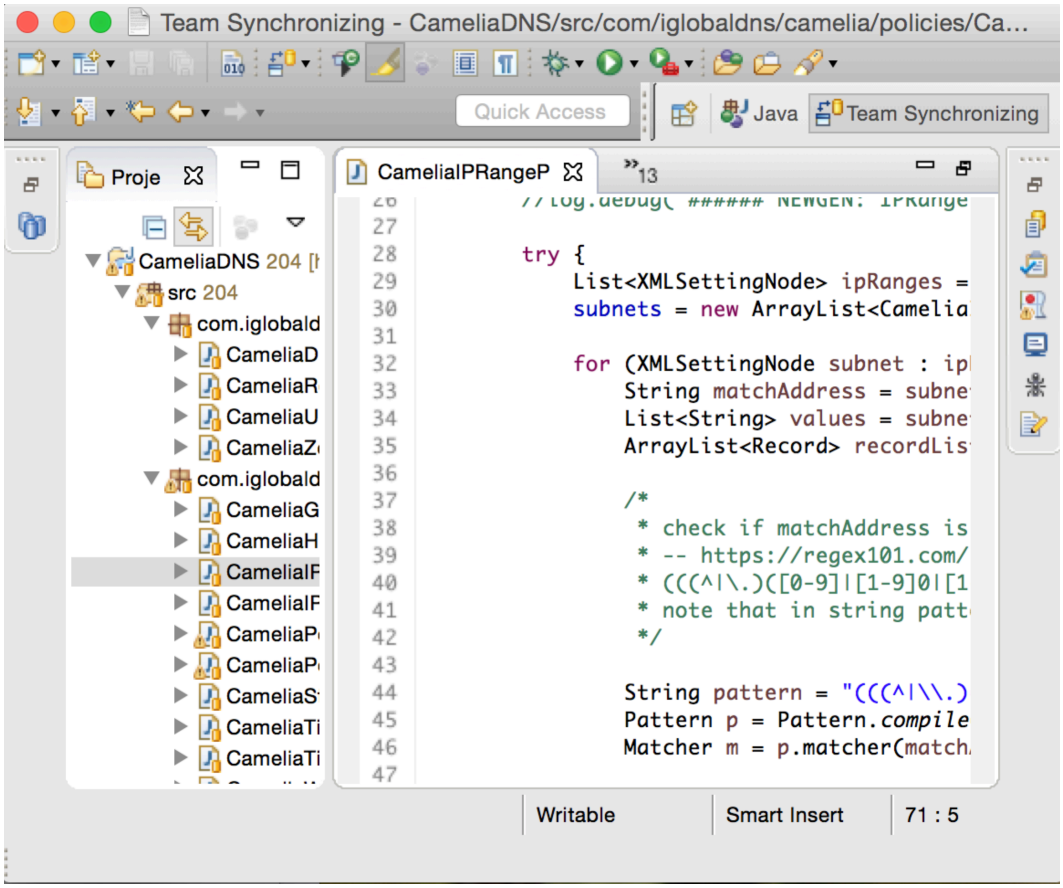
També primordial la tasca de SVN per revisionar Puppet ja que és molt interessant per veure l'evolució dels manifests i per entendre com han canviat els sistemes des del primer dia que es van instal·lar fins el moment actual.

Les versions usades han estat la 1.8.8 en aquest ultima fase del PFC. Tant per client com per servidor.

Eclipse

Competeix en importància amb Subversion i denota una vegada més la importància de Java en el projecte. Deixant de banda el haver programat en aquest llenguatge de programació Eclipse ha estat una eina vital per desenvolupar el codi i compilar-lo, tant al començament de tot com fins al final.

Aquesta pantalla m'ha acompanyat durant moltes hores en aquest PFC



Integrat amb Subversion per pujar els canvis i veure també les diferències existents.

Versions fetes servir al 2010:

- Eclipse IDE for Java Developers Helios Service Release 1
- SVN Connector: SVNKit 1.3.3

Versions fetes servir ara:

- Eclipse IDE for Java Developers Mars Release (4.5.0)
- SVN Connector: SVNKit 1.8.10

Es va haver de migrar de les versions antigues a les noves, amb el que implicava en quant a canvi d'entorn: sistema operatiu (de Windows 7 a MacOS 10.10), versions de Java (1.8 i 1.7 instal·lades al sistema ja que són les més modernes).

Ubuntu

Sistema Operatiu escollit en la fase final per desplegar la infraestructura.

La versió feta servir en aquest moment és la LTS que ofereix Ubuntu i per tant també Amazon, en versió 64bits.

Versió utilitzada: aprofitant que *Facter* està instal·lat a la plataforma...

```
ubuntu@vcs1 ~ $ facter lsbdistdescription
Ubuntu 14.04.3 LTS
```

Bash

Bash sempre està present quan es tracta d'administrar sistemes operatius Linux. Des de l'script d'arrencada de CameliaDNS, passant pel backup de Subversion fins la sincronització de codi compilat cap al master.

Versions: múltiples depenen de l'any i plataforma

Puppet

Fet servir en la fase finalíssima sobretot, només per administrar la infraestructura. És de vital importància per mantenir els sistemes actualitzats, i coherents entre si. És molt potent i al PFC només s'ha aplicat una pinzellada del que permet.

Al projecte ens ha permès poder desplegar canvis comuns a totes les màquines de manera controlada, actualitzar les versions de paquets importants com *Apache* o *Php* i ajudar-nos a comprendre que les configuracions de sistema s'han de definir, i controlar tant o més que el codi del software.

Versió feta servir: 3.4.3 (tant servidor com els clients)

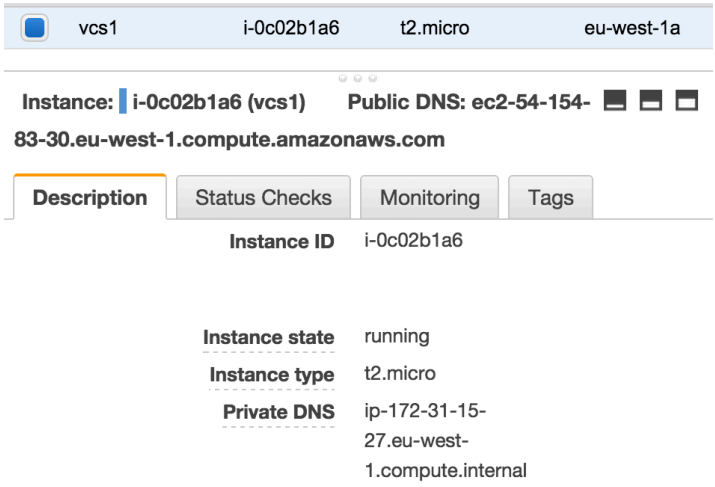
Ruby

Puppet i *Facter* corren en Ruby així que es mereix sortir a la llista de tecnologies usades. A més a més s'ha pogut treballar una mica amb aquest llenguatge de programació gràcies al haver desenvolupat *facts* personalitzats per *Facter*.

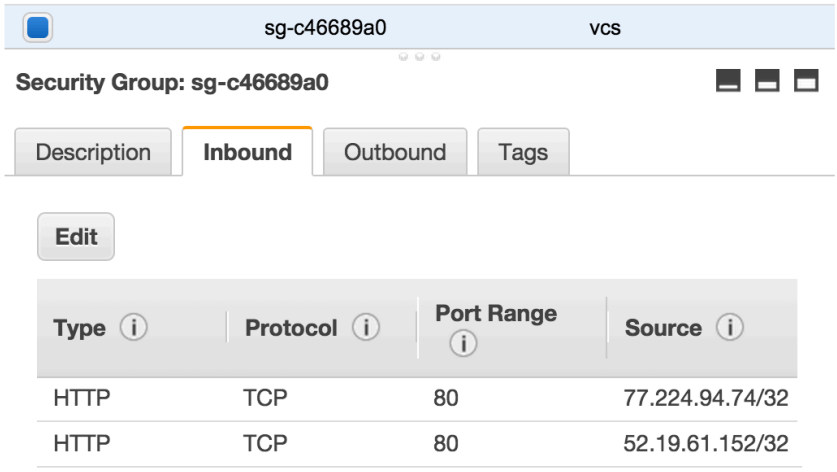
Versions utilitzades: 1.9.3

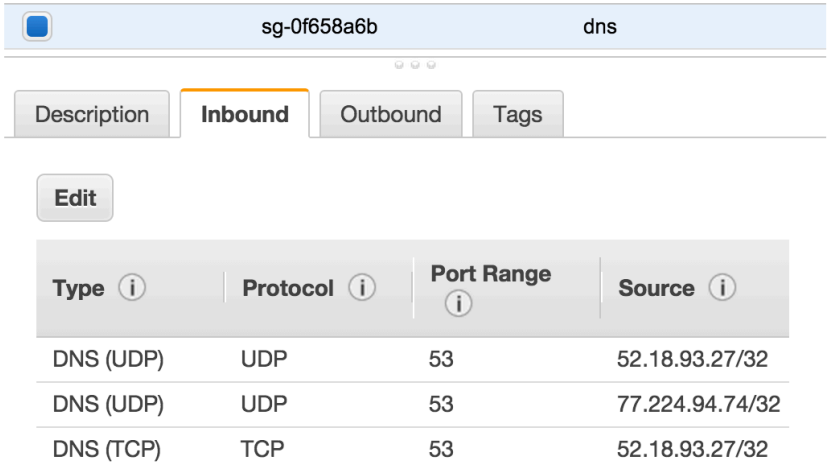
Amazon Web Services (AWS)

Gràcies a la flexibilitat d'Amazon i el seu servei AWS s'han pogut desplegar màquines que tenim accessibles worldwide, a un cost mínim.



S'han configurat security groups independents en funció del role de les instàncies per tal de juntament amb *Facter* i *Puppet* saber en quin tipus de màquina ens trobàvem i desplegar les configuracions adequades.





Apache

Servidor web de referència fet servir com a suport al PFC pel següents serveis:

- Subversion
- WebSVN
- Puppet

En tots els casos la versió utilitzada és la 2.4.7

WebSVN

Eina de visualització de Subversion basada en Php, molt útil per quan no es vol veure l'output cru del client de SVN, com aquest

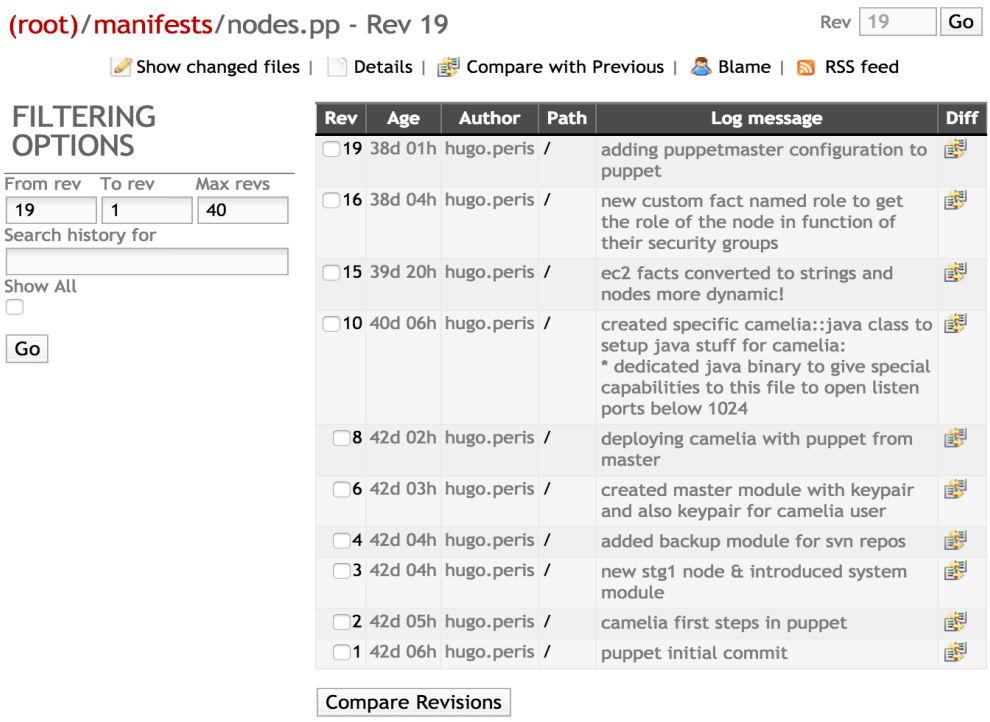
```
root@adm1 /etc/puppet/manifests # svn log nodes.pp
------------------------------------------------------------------------
r19 | hugo.peris | 2015-08-21 14:20:42 +0000 (Fri, 21 Aug 2015) | 3 lines

adding puppetmaster configuration to puppet


------------------------------------------------------------------------
r16 | hugo.peris | 2015-08-21 11:37:48 +0000 (Fri, 21 Aug 2015) | 3 lines

new custom fact named role to get the role of the node in function of their security
groups
```

en canvi amb WebSVN es poden visualitzar els canvis així:



Conclusions

Errors comesos

De tots els errors comesos en l'execució del PFC el més gran va estar aturar el PFC i trigar tant de temps en tornar-lo a reprendre.

Pot ser la idea d'un PFC conjunt era molt ambiciosa per la càrrega de feina que implicava i el haver-se de sincronitzar amb altres persones, tenint en compte que tots teníem les nostres respectives feines i vides personals, ho feia molt complicat i inviable sent realistes. Però això no era excusa per haver-lo tingut tant de temps aturat.

Per la part tècnica trobo que hagués estat millor en comptes de treballar tant en local, en seguida pujar codi i fer releases continues a entorns de proves. No parlo de la fase col·laborativa exclusivament, sinó també en la fase en solitari en els últims mesos.

En comptes de desenvolupar sobretot en local esperant tenir una bona versió i llavors "sortir" a provar als servidors de proves corrent en Linux. Fer-ho de seguida, per entrar en la roda.

És possible que llavors, i enllaçant amb el d'abans, haguessin sortit idees per fer una petita maqueta de desenvolupament continu. Tenir una màquina més no hagués costat molt i es podria haver instal·lat un Jenkins que pugés paquets a un repositoris de tipus Nexus.

També és cert que s'ha d'intentar que l'abast del projecte no se'n vagi molt lluny perquè es pot caure al parany de que per voler fer moltes coses finalment no es completi cap.

Un altre error és el no haver documentat certes parts el suficient. Un simple comentari al codi pot ajudar molt a entendre com funciona. Al moment de recuperar la feina feta anys enrere vaig veure que no entenia gairebé res del que jo mateix havia programat.

Les parts del codi que tenien algun comentari, per simple que fos, sobre la funció que realitzava una línia, o inclús alguna frase explicant el funcionament a grans trets del que es pretenia amb el mètode ajudaven moltíssim.

Però no eren la constant al codi, la majoria de codi no estava així i li vaig haver de dedicar molt de temps a descobrir com funcionava la versió 0 de CameliaDNS.

Trobo que a més és un error en el que he tornat a caure de nou perquè no he pogut dedicar-li tot el temps que m'hagués agradat a afegir comentaris. Així que en cas de reprendre l'evolució del codi en certs punts podria passar el mateix, ja que hi ha parts del software que funcionen, i ho fan be, però sense una explicació quan la idea marxa del cap costa més de veure.

I enllaçant amb el temps, un altre error es haver planificat malament l'esforç d'algunes tasques. Pensar que es podia fer una funcionalitat des de cero en 4 hores no era una bona planificació. I pensar que muntar Puppet per pocs nodes és 0 inversió de temps perquè ho tens per la mà també és un error. Sempre surt alguna cosa que et pot enrederir.

Llissons apresses

Com a llisó personal he aprés sobretot que no s'ha de trigar tant en realitzar un projecte d'aquest tipus. Primer per la implicació que té en la teva vida acadèmica. I segon perquè no s'han de deixar coses a mitges, d'una manera o una altra s'han d'acabar.

A nivell tècnic treballant en el PFC he aprés Java, és indubtable, no soc un programador expert però em defenc. Al reprendre el PFC aquest any em vaig adonar que encara recordava moltes coses.

A la meva vida professional no soc programador, així que codi Java no el toco directament. Administro serveis i eines que corren en Java però em centro amb la part de sistema, en com es comporta la JVM, i res en desenvolupar en Java. Així que es pot dir que he aprés Java gràcies al PFC. Ho demostra el fet de que com dic anys després d'haver deixat el projecte em sentia còmode en Java.

Una cosa que he vist és que quan s'està fent un projecte d'aquests tipus que per una banda s'ha de programar molt, i per l'altre encara que no és una exigència s'ha de muntar una part de infraestructura, està bé no focalitzar-se molt durant moltes hores, o dies seguits, en el mateix tipus de tasca.

Si es fa un canvi de context controlat a un tasca d'un altra tipus resulta bastant refrescant mentalment i ajuda a aclarir idees.

No dic de fer context-switching cada 10 minuts però si que si s'ha arribat a un punt on una tasca, com podria ser per exemple programar una funcionalitat nova, no acaba de sortir i s'està bloquejat, igual es bo deixar aquesta tasca apartada unes hores per avançar en una altra tipus de tasca nova, diferent, més motivadora com pot ser instal·lar un servidor.

Millores

Coses que es podrien millorar en CameliaDNS per exemple és el sistema de loggeig d'errors i debug. No és molt consistent entre el tipus de missatge que retorna quan hi ha un error, o quan ho fa. Faria falta una repassada a fons, i segurament definir un fitxer a part amb log4j per les classes de CameliaDNS.

Una petita infraestructura de desenvolupament continu i no fer tant desenvolupament i compilació local. Seria bo fer-ho per tenir una plataforma més actualitzada i més constantment. A més que tenir uns repositoris de software amb Nexus per les llibreries Java seria molt productiu.

Enllaçant amb això en comptes de fer deploys de codi via Puppet havent pujat abans el codi manualment al servidor master, aprofitar Nexus per fer el deploy d'aquest de codi. I segurament independitzar Puppet del deploy de software ja que no és una eina realment pensada per això, és pensada per canvis de configuració de sistema.

Tot i així si que es podria continuar fent servir Puppet si en el procés de release de CameliaDNS s'inclogués una part de paquetització del software en .rpm .deb o el que més agradés. D'aquesta manera els paquets de software es podrien pujar a un repositori local i instal·lar-los per Puppet.

Com a funcionalitats pel servei de DNS és inqüestionable que falta una que es va haver de treure dels objectius en la represa: la monitorització del hosts a retornar. Seria una bona funcionalitat a desenvolupar, sempre mantenint la idea de que no fos el propi CameliaDNS qui monitoritzés sinó que ho fes un altre procés que alimentés a CameliaDNS.

També es troba a faltar la combinació de polítiques. Per fer-ho simple la primera iteració consistia en fer una única política per CameliaRecord. Però tenia al cap si hi havia temps la possibilitat de combinar varies.

Per exemple segons la hora del dia tenir una resolució amb GEO Localització, i a unes altres hores tenir-la estàtica.

Seria una funcionalitat que transformant les classes i mètodes a un model recursiu és factible de fer.

Referències

- http://www.unlogic.se/projects/eagledns
- http://www.dnsjava.org/
- http://dev.maxmind.com/geoip/legacy/geolite/
- http://projects.puppetlabs.com/projects/1/wiki/Using_Passenger
- https://raw.githubusercontent.com/puppetlabs/puppet/stable/ext/rack/config.ru
- http://www.bind9.net/rfc
- https://regex101.com/r/pJ7bV5/2